I implemented the task using the following methods in the given order:

1. **Image Pre-Processing:**

   Denoising-

   Double Thresholding:

   The given depth image is not perfect, some edges may not actually be edges and there is some noise in the image. To overcome that, I have used double thresholding. It sets two thresholds, a high and a low threshold.

   In my algorithm, I retained the pixel values between this threshold and the rest are either set to a minima or a maxima(pixel too far or too close). Pixels with a high value are most likely to be edges. For example, you might choose the high threshold to be 4, this means that all pixels with a value larger than 4 will be a strong edge and are set to 255.

   I chose low threshold as 1, this means that all pixels less than it is not an edge and are set to 0. The values in between 0.3 and 0.7 would be weak edges, in other words, we do not know if these are actual edges or not edges at all. Step 6 will explain how we can determine which weak edge is an actual edge.

   Blurring:

   The current pixel values in the image between hard boundaries. To smoothen the edge values and get a more accurate detection we can apply methods such as Gaussian Blur, Median Blur.

   Median Blur takes median of all the pixels under kernel area and central element is replaced with this median value.

   Gaussian Blur effectively does an elementwise additive multiplication with a kernel to convert the given pixel map distribution to fit a curve defined by given standard deviation. Since our previous step of thresholding, the median blur made more sense to me as an intuitive salt-pepper noise that are sharp and sudden changes in pixel value than a uniformly distributed noise.

2. **Canny edge detection:** To find out the edges in the image, I have used opencv implementation of canny edge detector: `cv2.canny(image, lower, upper)`. The Canny edge detection algorithm can be broken down into 5 steps:

   Step 1: Smooth the image using a Gaussian filter to remove high frequency noise.

   Step 2: Compute the gradient intensity representations of the image.

   Step 3: Apply non-maximum suppression to remove “false” responses to to edge detection.

   Step 4: Apply thresholding using a lower and upper boundary on the gradient values.

   Step 5: Track edges using hysteresis by suppressing weak edges that are not connected to strong edges.

   The main problem becomes determining the lower and upper threshold values. To tackle that, I have computed the median of the single channel pixel intensities and used it to calculate the threshold values.

   Boosting Edges: To boost the edges, I have used the two most basic morphological image processing operations: *Dilation and Erosion*. Dilation adds an extra layer of pixels on a structure whereas erosion basically strips out the outermost layer of pixels in a structure.

3. **Contour Detection and Bounding Box Localization:** For my task , I am interested in the coordinates of the human. So, I have cropped the image to localize the object of interest. Having got the final region of interest, my next task is to determine the contour pixels. For this, I have used the opencv implementation of contour detection:
   `cv2.findContours(cropped, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)`
   Having found the contours, we want to find the bounding box as is the norm for generalization across object detection algorithms and for ease of calculating the distance from shelf and wall.
   The contours at this point are in the form of pairs of points. We can use them first to find the approx polygon as a closed boundary for all these point pairs.
   The function `cv2.approxPolyDP` approximates a curve or a polygon with another curve/polygon with less vertices so that the distance between them is less or equal to the specified precision.
   This gives us a closed polygon again as pairs of points enclosing the closed contour of the human. For getting the bounding box, next I have used `cv2.boundingRect` which returns the minimal up-right bounding rectangle for the specified point set or non-zero pixels of gray-scale image.

4. **Clearance distance:** After determining the coordinates of the bounding box, I have calculated the left and right clearance distances for the avoidance maneuver by the robot. In cases where the left clearance distance is greater than the right clearance distance, the robot is asked to move left otherwise move right.

5. **Next steps for my perception system:**

   Generalizing this algorithm would be subjective to the scale of deployment and the amount of training data.
   A large enough dataset and a requirement for inferences across a diverse distribution would be best suited by deep learning object detection algorithms like Yolo, ssd, Faster Rcnn. Yolo proves to be useful for really fast real time prediction and FRCnn proves to be more accurate though a bit slower. One could also think of designing a deep learning model based on an objection detection architecture but a modified cost function which penalizes distance between right/left edge pixels with pixels of bounding box to maximize distance as predicted obstruction free path.
   For smaller datasets, more intricate image processing and slam techniques can be used. Optimizing the current steps in my proposed solution like noise removal, edge detection, region of interest cropping should be focused on. Extra pre-processing steps like non-maximum suppression and or use of anchor boxes would help.